20 | 不同文件混在一起，怎么快速分类?

2021-03-27 尹会生

Python自动化办公实战课 进入课程 >

讲述：尹会生

时长 21:44 大小 19.92M

你好，我是尹会生。

今天我们的内容要从一碗香喷喷的蛋炒饭开始。要想做一份传说中的蛋炒饭，肯定要放胡萝卜、黄瓜、火腿肠还有葱花等好多种类的食材。

这是不是像你的桌面一样，为了完成某一项目，需要将音频、视频、文档、图片等各种格式组合在一起。但你在你完成了项目之后，想要将它们进行整理的时候，会发现各类文件堆满了桌面，要想从桌面找都某个文件就像从蛋炒饭里将所有的葱花挑出来一样困难。

对于这种情况，我们可以采用 Python 按照扩展名，分门别类地整理到不同的目录，方法虽然找到了，但是在你动手写代码的时候发现也不容易，就像从蛋炒饭中把鸡蛋、米饭、胡萝卜丁、火腿肠等食材挑出来，分类型放在不同的盘子中。这无疑会让你非常头痛。

所以在今天这节课中，我就带你来学习一下，怎么用我们之前学习过的自定义函数、队列，来实现按照扩展名对文件的自动分类。

批量分类的方法与思路

在带你学习代码之前，我要先为你讲解一下解决这类问题的思路，因为像自动分类这种场景，可以被拆解成判断类型逻辑和移动逻辑，而这两个逻辑有前后顺序，还有前后依赖关系。这一大类问题，你在工作中会经常遇到，当你学会了这类问题的解决思路之后，再遇到同类问题，就能非常容易的想到处理逻辑，那再通过代码将你的思路实现出来，也就不在话下了。

要想实现自动分类，就要设计好**分类的规则 ****，以及**按照规则对每一个文件分类的**通用模式**。我们先来学习设计分类规则。

怎样设计合理的数据类型

分类规则是指将扩展名和要移动的目录建立对应关系，而想要保存对应关系，就必须设计一个合理的数据类型，既能保存这种对应关系，又不能太复杂，为自己编码带来困扰。由此来看，分类规则的核心就是设计合理的数据类型。

这么说，你可能有点难以理解。我先把代码提供给你，然后我来带着你分析，我们为什么要设计数据类型。代码比较长，为了让你有更好的学习效果，我把分类前和分类后的文件目录结构都提供给你。通过对照分析分类前后文件所在的目录，帮你理解自动分类代码的实现思路。

分类前的目录和文件结构如下：

复制代码

```
$ ls -R files
dir1 a.mp4 c.rm d.avi b.mp3

files/dir1:
aa.exe  bb.bat
```

分类后的目录和文件结构如下：

复制代码

```
$ ls -R files
dir1  execute  movie  music

files/dir1:

files/execute:
aa.exe  bb.bat

files/movie:
a.mp4  c.rm  d.avi

files/music:
b.mp3
```

对比分类前后的目录和文件结构，可以看到，我并没有把每一种扩展名保存在一个独立的文件夹中，而是把这些文件按照音乐、视频和可执行文件的方式进行了分类，然后再把同一类型的文件放在相同目录中。

这样的实现方式为工作中查找文件带来了便利，但是不可避免地，会增加我们编码工作的复杂度，因为你不能通过循环遍历一次文件来实现分类了。

我这么表述，你可能还不太理解它的难度具体在哪里，我们还是回到蛋炒饭的例子。如果把每个扩展名都放在一个目录中，就类似把蛋炒饭中的每种原材料都放在一个碗里。你只要准备和原材料类型相同数量的碗，去分类就好了。

而分类方式如果变成了只有三个碗，我们此时需要把材料要把主食、素菜、荤菜分别放在三个碗中，那你在遍历蛋炒饭的时候，就需要二次分类。

对于这样的需求，你在编写代码前需要设计合理的数据类型，把碗的数量和蛋炒饭的原料对应关系事先确定好。而确定的这一对应关系在编程语言中就被称作设计数据类型。

那怎样来设计合理的数据类型呢？让我们来看看文件自动分类中的“碗和原材料”。

文件自动分类功能中的“碗”是多个文件夹。但是在 Python 中，表示多个文件夹的时候，我们会采用字符串，方便和文件夹名称建立对应关系。而且你还可以通过创建文件夹的库，把字符串作为文件夹名字，实现字符串到文件夹的对应关系。

文件自动分类功能中的“原材料”是扩展名。扩展名也要使用字符串类型。那么每组文件夹到扩展名对应关系都是一个字符串对应多个字符串，相信你一定想到了这种对应关系应该使用字典保存它们的映射关系了吧，那作为字典值的多个扩展名，由于在运行程序前指定好就不会再修改了，所以我将扩展名字符串组成元组，将元组作为字典的值。

那么根据我们的分析，我把扩展名和文件类型定义为如下字典，如果你的工作场景使用到了更多的扩展名，或者使用了和我不同的分类，也可以修改这个字典。

复制代码

```
# 定义文件类型和它的扩展名
file_type = {
    "music": ("mp3", "wav"),
    "movie": ("mp4", "rmvb", "rm", "avi"),
    "execute": ("exe", "bat")
}
```

通过刚才对字典的定义，我们给扩展名自动分类制定好了分类的规则。那接下来我们继续设计程序，按照这一分类规则，进行文件的读取和分类。

怎样设计生产者消费者模式

文件的读取和分类是两个不同的功能，你在编写代码时可以把它们编写成两个不同的函数。

但是由于今天的程序比以往要复杂，所以实现这两个函数的思路也会比较多。比如：

> 你可以把所有的文件全部读取之后，再按照分类规则移动到新的目录；
>
> 也可以读取一个紧接着将它移动到一个新的文件夹里；
>
> 当然还可以读取一部分文件之后，将一部分文件进行移动，然后继续读取第二批文件。

到底选择哪种方案才是最佳实践呢？

在这种情况下，你最希望的是能够向有丰富开发经验的开发人员请教，看他是怎么实现类似需求的，然后按照他的实现逻辑来编写你的代码。这也是专业软件开发人员面对这一问题时的通常做法：去寻找和当前场景相似的“设计模式”。因为设计模式是众多软件开发

人员经过相当长的时间，对某一场景进行大量试错总结出来的经验。那我们可以利用它，来解决我们当前场景下的问题。

我们当前的场景刚好和设计模式中的“生产者消费者模式”比较吻合。生产者消费者模式的描述是这样的：有两个进程共用一个缓冲区，两个进程分别是生产数据和消费数据的。而缓冲区，用于存放生产进程产生的数据，并让消费进程从缓冲区读取数据进行消费。

使用生产者消费者模式刚好能解决文件读取和文件分离的逻辑。我把读取当前文件名称和路径函数作为生产者，把分类和移动文件的逻辑作为消费者。在生产者消费者中间，我再使用队列作为它们中间的缓冲区。

可以看到，使用生产消费者模式，我主要是增加了一个队列，而不是从生产者直接把数据交给消费者。这样做主要有三个好处：

1. 如果生产者比消费者快，可以把多余的生产数据放在缓冲区中，确保生产者可以继续生产数据。
2. 如果生产者比消费者慢，消费者处理完缓冲区中所有数据后，会自动进入到阻塞状态，等待继续处理任务。
3. 缓冲区会被设置为一定的大小，当生产者的速度远远超过消费者，生产者数据填满缓冲区后，生产者也会进入到阻塞状态，直到缓冲区中的数据被消费后，生产者才可以继续写入。而当消费性能不足时，可以等待消费者运行，减少生产者和消费者在进度上相互依赖的情况。

通过分析我们发现，可以采用生产者消费者模式来编写文件的读取和分类代码。

考虑到是你初次接触设计模式，为了不让你产生较大的学习心理负担，我把其中的多线程并发访问缓冲区简化成单线程版本，这样你能根据代码的执行过程，先学会简单的生产者和消费者模式。

在分类规则的“file_type”字典之后，我增加了以下代码，实现了单线程版本的生产者消费者设计模式。如下：

复制代码

```
1 from queue import Queue
```

```
# 建立新的文件夹
make_new_dir(source_dir, file_type)

# 定义一个用于记录扩展名放在指定目录的队列
filename_q = Queue()

# 遍历目录并存入队列
write_to_q(source_dir, filename_q)

# 将队列的文件名分类并写入新的文件夹
classify_from_q(filename_q, file_type)
```

上面的代码实现了从定义队列到文件处理的完整函数调用。在后续（第 22 节、28 节）我为你讲完面向对象、类和多线程后，我会带你再实现一个多线程版本的生产者消费者模型，让你完全掌握这一设计模式，并应用到更多的场景中。

在确定了分类规则用到的数据模型，以及分类流程用到的设计模式之后，接下来就到了具体实现代码的环节了。

在生产者消费模式下，我通过定义三个函数来分别实现三个功能，如下：

定义函数 make_new_dir()，实现新建分类文件夹的功能；

定义函数 write_to_q()，实现写入当前文件路径到队列的功能；

定义函数 classify_from_q()，实现把队列中的文件分类并移动的功能。

接下来，我带你依次学习一下它们各自的实现代码。

如何实现分类

要想实现分类，首先要先创建分类需要的文件夹。这里需要注意，创建文件夹的操作要在批量分类前完成，否则在每次移动文件前，你还得对要移动的文件夹进行判断，这会影响程序的运行效率。

我们来看一下怎样利用分类规则的字典“file_type”，以及 make_new_dir() 函数来批量分类文件夹。

如何建立分类文件夹

批量建立文件夹操作的前提是建立哪几个文件夹，以及在哪个目录下建立它。基于这样的考虑，我为 make_new_dir() 函数增加了两个参数：

1. 使用 dir 指定建立文件夹的目录；
2. 使用 type_dir 指定按照哪个字典建立。

而建立文件夹，可以使用我们学习过的 os 模块，通过 os.mkdirs() 函数建立一个新的文件夹。代码如下：

复制代码

```
import os
# 定义文件类型和它的扩展名
file_type = {
    "music": ("mp3", "wav"),
    "movie": ("mp4", "rmvb", "rm", "avi"),
    "execute": ("exe", "bat")
}

source_dir = "/Users/user1/Desktop/files

def make_new_dir(dir, type_dir):
    for td in type_dir:
        new_td = os.path.join(dir, td)
        if not os.path.isdir(new_td):
            os.makedirs(new_td)

# 建立新的文件夹
make_new_dir(source_dir, file_type)
```

这段代码把字典的 key 作为文件夹名称，通过遍历字典来批量创建文件夹。这里还有两个你需要注意的技巧。

第一个是文件路径的拼接。代码中要新建的文件夹路径，是由“source_dir”和遍历字典得到的“字典的 key”两部分连接组成的。如果你使用字符串的连接函数“join()”函数来连接这两部分，你需要增加路径连接符号"/"，而如果你的操作系统从 mac 换成 windows，则需要使用反斜线""，这时候你就要再修改代码，把斜线改为正确的路径分隔符。

因此我采用了“os.path.join()”函数，这个函数会自动判断操作系统并增加斜线"/"，它还避免了你为已经有“/”的路径重复添加的问题。

另一个小技巧是判断目录是否存在。我在创建目录前，使用了 os.path.isdir() 函数，判断了目录是否存在，这样做的好处是避免重复创建目录。

另外，我还想教给你和它功能相近的两个函数，它们分别是 os.path.isfile() 和 os.path.isexist()。

前者用来判断该路径是否存在，并且是否是同一个文件类型。

后者用来判断路径是否存在，并且这个路径可以是文件也可以是目录。

结合代码中出现的 isdir() 函数，你就可以对一个目录到底是文件还是目录，以及是否存在进行判断了。

创建目录之后，我们就要开始对当前的文件进行遍历，并存入缓冲区中。

怎样遍历目录并写入队列

我先把遍历目录的代码写在下面，然后再为你详细讲解它。

复制代码

```
from queue import Queue

# 遍历目录并存入队列
def write_to_q(path_to_write, q: Queue):
    for full_path, dirs, files in os.walk(path_to_write):
        # 如果目录下没有文件，就跳过该目录
        if not files:
            continue
        else:
            q.put(f"{full_path}::{files}")

#########
source_dir = "/Users/user1/Desktop/files

# 定义一个用于记录扩展名放在指定目录的队列
filename_q = Queue()

# 遍历目录并存入队列
write_to_q(source_dir, filename_q)
```

这段代码实现了定义队列，并把指定目录下所有的文件名称和路径写入到队列中的功能。在这里有两个关键的知识点需要你掌握，它们分别是如何遍历目录，以及如何写入队列。

先来看如何遍历目录的函数。它在代码的第 5 行，叫做 os.walk() 函数，和之前我们学习过的 pathlib() 函数一样，都能实现对目录的遍历，但是它的返回值值得你学习一下。

我使用 for 循环遍历 walk() 函时，分别使用了 full_path、dirs 和 files 三个变量，因此 walk() 函数的返回值有三个。这三个变量分别对应每次遍历的文件的完整路径、文件所在的目录，以及该目录下所有文件名称的列表。

你可以根据你的工作场景灵活组合这三个变量，由于我在移动的场景需要文件的完整路径和文件名，所以我只使用了第一个参数 full_path 和第三个参数 files。

此外，我在实现遍历时，也像创建目录一样增加了容错。如果某一目录下没有文件，就不需要对该目录进行移动了，所以我使用了“if not files” 来判断 files 列表的值。

由于我增加了 not 关键字，if 的判断条件就从列表中包含文件，变成了列表中没包含任何一个文件。当条件成立时，则执行 continue 语句，跳过当前这次循环。而 else 语句中，是当 files 列表中包含了文件名称的处理过程，在这种情况下，我会将文件的完整路径和该路径下的文件列表放到缓冲区中。

在当前代码，我把队列这一数据类型作为缓冲区，它和我们之前学习过的多进程通信的队列功能和用法完全相同，区别则是我们导入的库名称不同。

要想把对象存入队列，可以使用 put() 函数。从队列取出数据，则可以使用 get() 函数。我把循环遍历得到的路径和文件名称均使用了 put() 函数存放到队列中，实现了生产者这一角色。

接下来，我们来学习消费者这一角色实现的代码，学习如何实现分类并将文件移动到新的文件夹的。

分类并移动到新的文件夹

同样的，我先把代码写在下面，然后再为你详细分析如何实现从队列取出文件名并进行分类的功能。

复制代码

```
# 移动文件到新的目录
def move_to_newdir(filename_withext, file_in_path, type_to_newpath):
    # 取得文件的扩展名
    filename_withext = filename_withext.strip(" \'")
    ext = filename_withext.split(".")[1]

    for new_path in type_to_newpath:
        if ext in type_to_newpath[new_path]:
            oldfile = os.path.join(file_in_path, filename_withext)
            newfile = os.path.join(source_dir, new_path, filename_withext)
            shutil.move(oldfile, newfile)

# 将队列的文件名分类并写入新的文件夹
def classify_from_q(q: Queue, type_to_classify):
    while not q.empty():
        # 从队列里取目录和文件名
        item = q.get()

        # 将路径和文件分开
        filepath, files = item.split("::")

        # 剔除文件名字符串出现的"[" "]",并用", "做分隔转换为列表
        files = files.strip("[]").split(",")
        # 对每个文件进行处理
        for filename in files:
            # 将文件移动到新的目录
            move_to_newdir(filename, filepath, type_to_classify)
```

在这段代码中，我实现了从队列取出文件名称和目录，并根据分类将文件移动到新的目录。

由于消费者的逻辑是从队列读取内容和移动文件两个功能组成的，所以我把消费者拆分成了两个函数进行编写：

1. classify_from_q() 函数，用来实现从队列读取文件列表，并遍历列表，得到每一个文件名称；
2. move_to_newdir() 函数，把文件名称、路径、分类规则作为参数，真正实现移动。

相应的，如果你在编写包含多个功能的程序时，也要尽量保持每个功能的独立性，把每一个功能尽量放在一个函数中，这样能有效提升你的代码的可读性。

这两个函数虽然比较长，但是大部分都是我们学过的内容，我想为你重点讲解一下第一次接触到的两个知识点，一个是 in 操作，一个是利用 shutil 库的 move() 函数实现的重命名。

in 操作叫做成员操作符，它能支持目前我们学习过的所有基础数据类型，用来判断一个值是否是列表、元组、字典等基础数据类型中的一员。如果这个值是基础类型的成员之一就会直接返回 True，如果不是成员之一返回的就是 False。有了 in 操作符，你就不用手动遍历基础数据类型，再使用“==”逐个去判断某个值和数据类型中的成员是否相等了。

我举个例子，你会更容易理解。我在代码中使用了这样一行代码：“if ext in type_to_newpath[new_path]”：

“ext” 就是文件的扩展名，就像是“a.mp3”的扩展名是“mp3”；

“type_to_newpath[new_path]”是字典“type_to_newpath”中，以“new_path”作为 key 的值，就像是“type_to_newpath = { "music": ("mp3", "wav") }”的“("mp3", "wav")”。
我把变量改成具体的变量值，那这行代码就变成了下面的样子：

复制代码

```
1 "mp3" in ("mp3", "wav")
```

如果扩展名在元组中，那么 if 条件的返回结果就是 True，就可以进行文件的移动了，如果结果是 False 则从字典中继续取下一个 key，直到所有的 key 遍历完成之后，仍然没有匹配的扩展名，就把文件保持在原地，不做任何移动操作。

还有一个我们第一次接触到的函数是 shutil 库的 move() 函数，这个函数是直接对系统上的文件进行操作的，所以你需要注意移动以后的文件名不要和已有的文件名冲突，这样会导致重名覆盖已有的文件，从而丢失文件。因此在你没有十足的把握之前，建议你在移动前增加一个判断功能，判断移动的文件是否存在，如果存在则提示使用脚本的人此情况，或移动前将文件进行改名。

以上就是如何对混在一起的多个扩展名的文件，进行自动分类的完整过程。这节课的完整代码比较长，我一并贴在了下方，帮你理解多个函数之间的调用关系和执行顺序。

复制代码

```
import os
import shutil
from queue import Queue

# 建立新的目录
def make_new_dir(dir, type_dir):
    for td in type_dir:
        new_td = os.path.join(dir, td)
        if not os.path.isdir(new_td):
            os.makedirs(new_td)

# 遍历目录并存入队列
def write_to_q(path_to_write, q: Queue):
    for full_path, dirs, files in os.walk(path_to_write):
        # 如果目录下没有文件，就跳过该目录
        if not files:
            continue
        else:
            q.put(f"{full_path}::{files}")

# 移动文件到新的目录
def move_to_newdir(filename_withext, file_in_path, type_to_newpath):
    # 取得文件的扩展名
    filename_withext = filename_withext.strip(" \'")
    ext = filename_withext.split(".")[1]

    for new_path in type_to_newpath:
        if ext in type_to_newpath[new_path]:
            oldfile = os.path.join(file_in_path, filename_withext)
            newfile = os.path.join(source_dir, new_path, filename_withext)
            shutil.move(oldfile, newfile)

# 将队列的文件名分类并写入新的文件夹
def classify_from_q(q: Queue, type_to_classify):
    while not q.empty():
        item = q.get()

        # 将路径和文件分开
        filepath, files = item.split("::")

        files = files.strip("[]").split(",")
        # 对每个文件进行处理
        for filename in files:
            # 将文件移动到新的目录
            move_to_newdir(filename, filepath, type_to_classify)

```

```
if __name__ == "__main__":
    # 定义要对哪个目录进行文件扩展名分类
    source_dir = "/Users/edz/Desktop/files"

    # 定义文件类型和它的扩展名
    file_type = {
        "music": ("mp3", "wav"),
        "movie": ("mp4", "rmvb", "rm", "avi"),
        "execute": ("exe", "bat")
    }

    # 建立新的文件夹
    make_new_dir(source_dir, file_type)

    # 定义一个用于记录扩展名放在指定目录的队列
    filename_q = Queue()

    # 遍历目录并存入队列
    write_to_q(source_dir, filename_q)

    # 将队列的文件名分类并写入新的文件夹
    classify_from_q(filename_q, file_type)

```

小结

最后让我来为你做个总结，实现文件自动分类是目前我们编写代码量最多的一讲。面对功能复杂、代码量增多时，你就需要通过函数设计合理的功能封装，还要考虑如何使用参数进行函数的通信。

当你的多个函数之间的工作流程也可以进行多种组合时，你可以借助开发高手的代码经验 -- 设计模式，来实现工作逻辑上的函数组合。在本讲中我为你介绍的这种普遍应用于产品生产、销售的生产者消费者模式就是设计模式中最常用的一种。

希望你能在掌握如何使用 Python 提高工作效率的同时也能掌握设计模式、函数这些编写 Python 的思路。这样，你在面对更庞大的需求时，也会更快地设计出结构清晰、逻辑清楚的代码。高效编程也是高效办公的一部分！

思考题

我来为你留一道思考题，如果我按照文件的大小对文件分成三类，将“大于 1GB” “1GB 到 100MB” “小于 100MB”三类的文件名和大小，依次显示在屏幕上，你会怎样实现

呢?

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | http库：如何批量下载在线内容，解放鼠标（下）？

下一篇 21 | SQLite文本数据库：如何进行数据管理（上）？

精选留言 (2)

写留言

mark for luck
2021-03-29

尹老师，能够建个微信群，学习前面就发现很多基础问题卡住了

编辑回复: 课程简介页面可以入群哦

LDxy
2021-03-28

好像把所有的功能都可以封装为函数，然后调用就行了。那什么情况下要使用类呢？见到有些python代码里面使用了类

展开

作者回复: 这个问题非常好。相对于类，函数定义和调用更加简单。
但是他在封装和继承上边要比类弱很多。从封装上来说，一般是一个功能，封装为一个函数。那么如果一个比较复杂的功能里面还有子功能，通过函数就很难来表达这样的概念。这时候可以使用类来表达一个复杂的一大类功能，再通过类当中的方法来去表达这一大类当中的子功能。

在继承方面也是这样，如果。有其他的函数，只引用当前函数的某一部分功能的时候，你不能将函数里面的部分代码再拆开成更小的函数。而累就可以通过继承的方式，通过子类来继承父类中的某一方法。达到指引用类中某一段代码的目的。

所以呢，如果追求简单还是用函数，追求封装继承还是要用类。

1